
Rezumatul conținutului

Ce sunt Load Balancers? Un load balancer acționează ca un distribuitor de trafic, primind cereri de la utilizatori și distribuindu-le uniform pe mai multe servere. Scopul principal este de a preveni supraîncărcarea unui singur server, de a asigura disponibilitatea serviciilor și de a îmbunătăți performanța generală a aplicației

Beneficii cheie:

- **Scalabilitate:** Permite adăugarea sau eliminarea dinamică a serverelor în funcție de trafic.
- **Fiabilitate:** Asigură redundanța, redirecționând automat traficul de la un server care a picat către cele funcționale.
- **Performanță:** Reduce latența și îmbunătățește viteza de răspuns, distribuind sarcina

Tipuri de Load Balancers

Videoclipul diferențiază principalele tipuri:

- **Layer 4 vs. Layer 7:** Load balancers-urile de la **Layer 4** se bazează pe adrese IP și porturi, fiind foarte rapide. Cele de la **Layer 7**] sunt mai avansate și pot lua decizii de rutare pe baza datelor din anteturile HTTP, cum ar fi URL-uri sau tipul de conținut.
- **Hardware vs. Software:** Există soluții fizice (hardware) și soluții software care rulează pe servere standard.
- **Cloud-based:** Servicii gestionate oferite de furnizorii de cloud, care simplifică procesul de configurare.
- **Global Server Load Balancers (GSLB):** Distribuie traficul între servere aflate în diferite locații geografice pentru a reduce latența pentru utilizatorii din întreaga lume de lucru.

Algoritmi de distribuție

Echilibratoarele de sarcină folosesc diversi algoritmi pentru a distribui traficul:

- **Round Robin:** Distribuie cererile în mod secvențial, pe rând .
- **Weighted Round Robin:** Alocă mai multe cereri serverelor cu o capacitate mai mare .
- **Least Connections/Least Time:** Direcționează traficul către serverul cu cele mai puține conexiuni active sau cu cel mai scurt timp de răspuns .

Ce este un Load Balancer și de ce este important?

Un load balancer este o componentă de infrastructură care acționează ca un "polițist al traficului" pentru cererile de rețea. El primește cererile de la utilizatori și le distribuie inteligent către un grup de servere (un "server pool"). Fără un load balancer, un singur server ar putea fi copleșit de un număr mare de cereri, ceea ce ar duce la o performanță slabă sau chiar la blocarea serviciului.

Beneficiile cheie sunt:

- **Disponibilitate ridicată:** Dacă un server dintr-un grup se defectează, load balancer-ul detectează problema și redirecționează automat traficul către serverele funcționale, asigurând continuitatea serviciului (failover).
- **Scalabilitate:** Permite adăugarea sau eliminarea dinamică de servere pentru a face față fluctuațiilor de trafic, fără a afecta performanța.
- **Performanță îmbunătățită:** Distribuind uniform cererile, reduce latența și timpul de răspuns.
- **Mentenanță ușoară:** Permite scoaterea unui server din rotație pentru mentenanță sau actualizări, fără a întrerupe serviciul pentru utilizatori.

Tipuri de Load Balancers

Videoclipul prezintă o clasificare a echilibratoarelor de sarcină în funcție de tehnologie și de nivelul la care operează în modelul OSI:

- **Load Balancers Hardware vs. Software:**
    - **Hardware**: Sunt dispozitive fizice dedicate, optimizate pentru o performanță ridicată, dar cu un cost mai mare și o flexibilitate mai redusă.
    - **Software**: Aplicații care rulează pe servere standard, oferind o mai mare flexibilitate și un cost mai redus, fiind ușor de implementat și de scalat în medii virtuale și cloud.
- **Layer 4 vs. Layer 7:** Acesta este un aspect tehnic crucial:
    - **Load Balancers Layer 4 (Transport Layer)**: Operează la nivelul protocolului TCP/UDP. Ei iau decizii de rutare bazate pe adresele IP sursă/destinație și pe porturi. Sunt foarte rapizi și eficienți, dar mai puțin "inteligenți" în distribuirea traficului.
    - **Load Balancers Layer 7 (Application Layer)**: Operează la nivelul aplicației (HTTP/HTTPS). Aceștia pot inspecta conținutul pachetului de date, cum ar fi anteturile HTTP, URL-urile sau cookie-urile, pentru a lua decizii mai sofisticate. De exemplu, pot direcționa traficul către un anumit server în funcție de tipul de conținut solicitat (static vs. dinamic). Această capacitate le permite, de asemenea, să gestioneze terminarea SSL, descărcând serverele web de sarcina criptării/decriptării.
- **Global Server Load Balancers (GSLB)**: Acestea sunt soluții avansate care distribuie traficul nu doar între serverele dintr-un singur centru de date, ci între servere situate în mai multe locații geografice. Astfel, utilizatorii sunt direcționați către cel mai apropiat server, reducând latența și crescând reziliența la nivel global.

Algoritmi de distribuire a traficului

Videoclipul descrie modul în care load balancers-urile decid către ce server să direcționeze o cerere, folosind diverși algoritmi:

- **Round Robin**: Un algoritm simplu care distribuie cererile în mod secvențial către fiecare server din grup.
- **Weighted Round Robin**: O versiune mai avansată, în care serverele cu o capacitate mai mare (mai multă memorie RAM sau putere de procesare) primesc mai multe cereri.
- **Sticky Round Robin:** Asigură ca cererile de la un anumit utilizator să fie direcționate întotdeauna către același server, util pentru aplicațiile care mențin o stare a sesiunii.
- **Least Connections**: Redirecționează traficul către serverul care are cele mai puține conexiuni active la momentul respectiv.
- **Least Time:** Alege serverul cu cel mai scurt timp de răspuns.
- **IP/URL Hashing**: Folosește o funcție de hashing pentru a direcționa întotdeauna traficul de la o anumită adresă IP sau pentru un anumit URL către același server.

Un **load balancer** este un dispozitiv sau un serviciu care distribuie traficul de rețea între mai multe servere. Gândește-te la el ca la un "polițist de trafic" virtual: în loc să trimită toate solicitările către un singur server, care ar putea fi copleșit și s-ar bloca, le direcționează inteligent către serverele disponibile.

Acest proces se numește **load balancing** (echilibrarea sarcinii) și are rolul de a îmbunătăți performanța, fiabilitatea și scalabilitatea unei aplicații sau a unui site web.

Cum funcționează?

Un load balancer stă între utilizatori și grupul de servere. Când un utilizator face o solicitare (de exemplu, accesează o pagină web), solicitarea ajunge mai întâi la load balancer. Acesta folosește un algoritm specific pentru a decide care server din grup este cel mai potrivit să proceseze solicitarea, luând în considerare factori precum:

- Numărul de conexiuni active pe fiecare server.
- Timpul de răspuns al fiecărui server.
- Capacitatea generală a serverelor.

După ce a ales serverul, load balancer-ul direcționează solicitarea către el. Răspunsul serverului este apoi trimis înapoi la utilizator, tot prin intermediul load balancer-ului.

Tipuri de algoritmi și beneficii

Există mai multe tipuri de algoritmi de load balancing, de la cei simpli și statici, cum ar fi **Round Robin** (care trimite solicitările pe rând la fiecare server), la cei mai complecși și dinamici, cum ar fi **Least Connections** (care trimite solicitarea către serverul cu cele mai puține conexiuni active).

Folosirea unui load balancer oferă o serie de beneficii importante:

- **Disponibilitate ridicată și fiabilitate:** Dacă un server se defectează, load balancer-ul detectează automat problema și nu mai trimite trafic către el, asigurând astfel că site-ul sau aplicația rămân funcționale.
- **Scalabilitate:** Poți adăuga cu ușurință servere noi pentru a face față unui trafic crescut, iar load balancer-ul va începe automat să le folosească. La fel de simplu, poți elimina servere când traficul scade, optimizând costurile.
- **Performanță îmbunătățită:** Prin distribuirea uniformă a sarcinii, se reduce timpul de răspuns și se evită supraîncărcarea unui singur server, oferind o experiență mai bună utilizatorilor.

În ecosistemul Spring Boot, load balancing-ul (echilibrarea sarcinii) și Eureka sunt folosite împreună pentru a gestiona în mod dinamic microserviciile.

Eureka joacă rolul de **Service Registry** (registru de servicii), iar load balancing-ul este implementat la nivel de client, de obicei folosind **Spring Cloud LoadBalancer**.

---

1. Rolul Eureka (Service Discovery)

Eureka Server acționează ca o bază de date centralizată unde fiecare microserviciu (client) se înregistrează.

1. **Înregistrare**: Când un microserviciu pornește (de exemplu, un "service-produse"), acesta se înregistrează cu adresa sa (IP și port) la serverul Eureka.
2. **Heartbeat**: Fiecare serviciu înregistrat trimite periodic un semnal ("heartbeat") către Eureka pentru a-i semnala că este activ și sănătos.
3. **Descoperire**: Un alt microserviciu (de exemplu, un "service-comenzi") care are nevoie să comunice cu "service-produse" nu știe adresa exactă a acestuia. În loc să folosească o adresă codificată, el întreabă serverul Eureka "care sunt instanțele disponibile pentru 'service-produse'?"
4. **Lista de instanțe**: Eureka îi oferă serviciului-client o listă cu toate instanțele active ale "service-produse", cum ar fi `service-produse:8081`, `service-produse:8082` etc.

---

2. Rolul Load Balancing-ului (Spring Cloud LoadBalancer)

Aici intervine load balancing-ul la nivel de client. După ce clientul primește lista de instanțe disponibile de la Eureka, el alege un server din listă pentru a trimite solicitarea.

Acesta este un model de **client-side load balancing** (echilibrarea sarcinii pe partea clientului), spre deosebire de un load balancer tradițional (hardware sau server-side).

- **Funcționare**: O bibliotecă precum **Spring Cloud LoadBalancer** (succesorul lui Netflix Ribbon) este integrată în microserviciul-client. Când acesta vrea să facă o solicitare către "service-produse", Spring Cloud LoadBalancer preia cererea, alege o instanță din lista obținută de la Eureka (folosind un algoritm precum Round Robin) și o direcționează către adresa corectă.
- **Avantaj**: Acest model este foarte eficient în medii cloud sau containerizate, unde instanțele pot apărea și dispărea frecvent. Clientul este cel care "se ocupă" de distribuirea traficului, fără a avea nevoie de un dispozitiv extern.

În rezumat

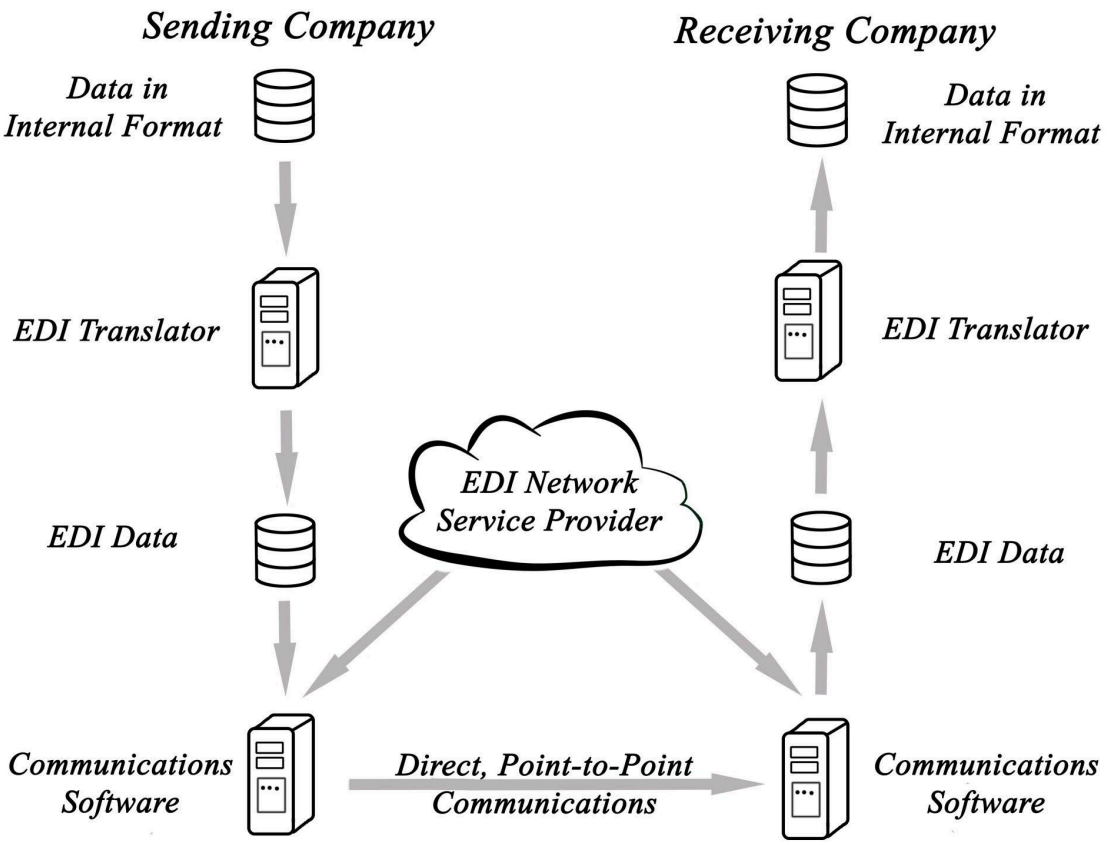
Eureka și Spring Cloud LoadBalancer funcționează împreună pentru a asigura un sistem de microservicii robust și scalabil:

1. **Eureka ➡️ Cunoaște** toate instanțele disponibile ale fiecărui serviciu.
2. **Spring Cloud LoadBalancer ➡️ Folosește** această cunoaștere pentru a distribui inteligent traficul către acele instanțe, direct de la client.

Implementarea unui load balancer într-un ecosistem Spring Boot se bazează pe principiile arhitecturii de microservicii, unde responsabilitatea echilibrării sarcinii este împărțită între serverul de servicii (Service Registry) și clientul care consumă aceste servicii.

Pe Partea de Server (Service Provider)

Partea de server se referă la microserviciul care furnizează o anumită funcționalitate (ex: un serviciu de produse).



Licențiată de Google

1. **Înregistrare în Service Registry**: Microserviciul trebuie să se înregistreze automat la un server de descoperire a serviciilor (cum ar fi **Eureka Server**). Când pornește, el își trimite numele, adresa IP și portul către Eureka. Această acțiune se numește **Service Registration**.
2. **Heartbeat**: Pentru a-și dovedi starea de sănătate și a rămâne vizibil pentru ceilalți, microserviciul trimite periodic un semnal "heartbeat" către Eureka. Dacă un serviciu nu mai trimite heartbeat-uri, Eureka îl marchează ca fiind inactiv.
3. **Configurare simplă**: Pentru a activa aceste funcționalități, este nevoie doar de o configurație minimă. De obicei, adaugi o dependență de `spring-cloud-starter-netflix-eureka-client` sau `spring-cloud-starter-consul-client` și anotezi clasa principală a aplicației cu `@EnableEurekaClient` sau `@EnableDiscoveryClient`.

Pe Partea de Client (Service Consumer)

Partea de client este microserviciul care are nevoie să consume un serviciu de la un alt microserviciu (ex: un serviciu de comenzi care apelează serviciul de produse).

1. **Descoperire Serviciu (Service Discovery)**: Clientul nu cunoaște adresa fizică a serviciului-țintă. El va interoga Service Registry-ul (Eureka) pentru a obține o listă a tuturor instanțelor disponibile ale acelui serviciu, folosind doar numele său logic.
2. **Load Balancing la Nivel de Client**: După ce obține lista de instanțe, clientul folosește un algoritm de load balancing (de obicei **Spring Cloud LoadBalancer**) pentru a alege o instanță din listă. Solicitarea este apoi trimisă direct către instanța aleasă.
3. **Integrare cu REST Clients**: Pentru a simplifica procesul, se folosesc clienți REST specializați, cum ar fi `RestTemplate` cu o adnotare `@LoadBalanced` sau, mai modern, `WebClient`. Acestea sunt capabile să rezolve numele serviciului în loc de o adresă URL fixă, datorită integrării cu Spring Cloud LoadBalancer.

Principii de Bază

- **Service Registration & Discovery**: Serviciile se înregistrează la un registru central (Eureka) și se pot găsi reciproc prin nume, nu prin adrese IP.
- **Client-Side Load Balancing**: Clientul este responsabil de distribuirea cererilor către instanțele disponibile, eliminând necesitatea unui load balancer extern.
- **Decuplare**: Fiecare microserviciu este independent. Poți adăuga sau elimina instanțe fără a modifica codul altor servicii, ceea ce asigură o scalabilitate ridicată.

În regulă. Să detaliem modul în care se implementează load balancing-ul în Spring Boot și principiile care stau la baza acestei arhitecturi, atât pentru client, cât și pentru server.

Partea de Server (Service Provider)

Un microserviciu care oferă o anumită funcționalitate trebuie să se asigure că este "descoperit" de alte servicii. Acest lucru se face printr-un mecanism de **Service Registration**.

- **1. Rolul Service Registry-ului (Eureka Server)**: Eureka Server este un registru centralizat. Fiecare instanță a unui microserviciu (de exemplu, un serviciu de produse care rulează pe porturile `8081`, `8082`, etc.) se înregistrează la Eureka cu un nume logic (de exemplu, `service-produse`). Aceasta elimină necesitatea de a folosi adrese IP și porturi hardcodate.
- **2. Cum funcționează înregistrarea**:
    1. **Activare**: Într-o aplicație Spring Boot, adaugi dependența de Eureka Client (`spring-cloud-starter-netflix-eureka-client`).
    2. **Configurare**: În fișierul `application.yml` sau `application.properties`, definești numele aplicației (`spring.application.name=service-produse`) și adresa serverului Eureka (`eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka`).
    3. **Adnotare**: Adaugi adnotarea `@EnableEurekaClient` pe clasa principală a aplicației Spring Boot.

După aceste etape, aplicația, la pornire, se va înregistra automat la Eureka.

- **3. Health Checks (verificări de sănătate)**: Serviciul înregistrat trimite periodic un semnal numit **"heartbeat"** către Eureka Server. Acest semnal este o confirmare că serviciul este activ și funcționează corect. Dacă Eureka nu primește heartbeat-ul de la o anumită instanță pentru o perioadă definită de timp, o consideră inactivă și o elimină din lista de instanțe disponibile. Această funcționalitate asigură **reziliența** sistemului.

Partea de Client (Service Consumer)

Clientul este aplicația care are nevoie să interacționeze cu un alt serviciu. Deoarece adresele serverelor se schimbă dinamic, clientul nu poate să le cunoască în avans.

- **1. Principiul Service Discovery**: Clientul (de exemplu, un serviciu de comenzi) interoghează serverul Eureka pentru a obține adresele IP și porturile tuturor instanțelor disponibile ale serviciului pe care vrea să-l apeleze (`service-produse`). Eureka returnează o listă de instanțe active.
- **2. Load Balancing la Nivel de Client (Client-Side Load Balancing)**: Odată ce clientul are lista de instanțe, el are nevoie de un mecanism care să aleagă pe care dintre ele să o apeleze. Aici intră în joc **Spring Cloud LoadBalancer**.
    - **Cum funcționează**:
        1. Clientul are integrată biblioteca Spring Cloud LoadBalancer.
        2. Când o cerere HTTP este inițiată (de exemplu, folosind `WebClient`), LoadBalancer-ul interceptează cererea.
        3. Folosește un algoritm de load balancing (cum ar fi **Round Robin** sau **Least Connections**) pentru a selecta o instanță dintr-o listă de instanțe sănătoase primită de la Eureka.
        4. În loc să trimită cererea la o adresă URL fixă, o trimite la adresa IP și portul instanței alese.
- **3. Integrarea cu clienții HTTP**: Pentru a face acest proces transparent pentru dezvoltator, Spring Cloud oferă un suport simplu:
    - **Cu `RestTemplate`**: Se adaugă adnotarea `@LoadBalanced` pe o instanță de `RestTemplate`. Aceasta îi spune lui Spring să injecteze un `LoadBalancerInterceptor` în clientul REST, care va prelua numele serviciului (ex: `http://service-produse/api/produse`) și îl va rezolva într-o adresă IP și port.
    - **Cu `WebClient`**: Aceeași logică se aplică și pentru `WebClient`, clientul HTTP reactiv modern.

Acest model de **client-side load balancing** este fundamental pentru arhitecturile de microservicii bazate pe Spring Cloud și Eureka, permițând scalarea orizontală și reziliența aplicațiilor.

`RestTemplate` este o clasă în Spring Framework folosită pentru a efectua solicitări HTTP de la client către server. Este o modalitate simplă și sincronă de a interacționa cu servicii RESTful.

Cum funcționează?

`RestTemplate` se bazează pe principii similare cu cele ale unui client web (cum ar fi un browser) care face o cerere către un server:

1. **Construire URL**: Specifici URL-ul către care vrei să trimiți solicitarea, de exemplu `http://api.exemplu.com/utilizatori/123`.
2. **Alegere Metodă HTTP**: Alegi metoda HTTP potrivită, cum ar fi `GET`, `POST`, `PUT`, `DELETE`.
3. **Trimitere Solicitare**: `RestTemplate` trimite solicitarea HTTP.
4. **Procesare Răspuns**: Când primește răspunsul de la server, `RestTemplate` poate converti automat corpul răspunsului (de exemplu, un JSON) într-un obiect Java.

Exemple de utilizare

Un exemplu simplu de cum poți folosi `RestTemplate` pentru a obține un utilizator de la un API arată cam așa:

Java
```
// Cream o instanță de RestTemplate
RestTemplate restTemplate = new RestTemplate();

// Definim URL-ul API-ului
String url = "https://api.exemplu.com/utilizatori/{id}";

// Apelăm metoda GET și convertim răspunsul în clasa Utilizator
Utilizator utilizator = restTemplate.getForObject(url, Utilizator.class, 123);
```

Caracteristici cheie

- **Sincron**: `RestTemplate` este un client HTTP blocant, ceea ce înseamnă că thread-ul care face solicitarea va aștepta până când primește un răspuns.
- **Integrare cu Spring**: Se integrează perfect în aplicațiile Spring Boot, fiind ușor de configurat și utilizat.
- **Conversie automată**: Poate converti automat tipurile de date comune (JSON, XML) în obiecte Java și invers, folosind un `HttpMessageConverter`.

De ce este considerat "legacy"?

Deși este încă folosit în multe proiecte, `RestTemplate` este considerat "legacy" de către echipa Spring. Motivul principal este natura sa **sincronă și blocantă**, care nu se potrivește bine cu arhitecturile moderne bazate pe programare reactivă și non-blocantă.

Alternativa recomandată este **`WebClient`**, care face parte din Spring WebFlux și suportă atât comunicare sincronă, cât și asincronă, oferind o performanță mai bună în medii cu trafic mare.

Pasul 7: Implementarea Resilienței cu Circuit Breaker

Vom folosi o tehnică numită **Circuit Breaker** (Întrerupător de circuit) pentru a gestiona eșecurile. Ideea este simplă: dacă un serviciu eșuează de un anumit număr de ori, circuit breaker-ul "deschide circuitul" și redirecționează apelurile ulterioare către o metodă alternativă (**fallback method**). După o perioadă, încearcă din nou să trimită apeluri către serviciul eșuat pentru a vedea dacă și-a revenit.

Vom implementa acest lucru în `Product Service` pentru a proteja apelurile către `User Service`.

Pasul 7: Implementarea Resilienței cu Circuit Breaker

Acest Canvas detaliază cum să adaugi Circuit Breaker (Întrerupător de circuit) la `Product Service` pentru a gestiona eșecurile în comunicarea cu `User Service`. Implementarea se bazează pe **Resilience4j**, o librărie modernă și eficientă.

1. **Dependențe Maven:** Am adăugat `spring-cloud-starter-circuitbreaker-resilience4j` și `spring-boot-starter-aop` în fișierul `pom.xml` al `product-service`. AOP (Aspect-Oriented Programming) este necesar pentru a permite adnotării `@CircuitBreaker` să funcționeze.
2. **Configurație:** Am inclus o configurație de bază pentru Resilience4j în `application.properties`, care stabilește cum ar trebui să se comporte circuit breaker-ul. De exemplu, `failureRateThreshold` (pragul de eșec) și `waitDurationInOpenState` (timpul de așteptare în starea deschisă).
3. **Metodă Fallback:** Vom crea o metodă alternativă (`fallback method`) care va fi apelată automat atunci când circuit breaker-ul este deschis. Aceasta va returna o valoare implicită sau un mesaj de eroare prietenos, în loc să lase cererea să eșueze complet. Această logică va fi adăugată în `ProductService.java` în următorul pas.

Acum, când `User Service` nu este disponibil, `Product Service` va răspunde cu o metodă fallback, prevenind blocarea aplicației.